CIT 5950 Recitation 2

Debugging, Structs and the Heap

Logistics

Check-in00

Due Monday January 23rd @ 10:00 am

Pre Semester Survey

Due Tuesday January 24th @ 11:59 pm

HW1 (Linked List & Hash Table)

Due Thursday January 26th @ 11:59 pm

Debugging

Debugging Overview

- **Debugging is a skill that you will need throughout your career**

- gdb (GNU Debugger) is a debugging tool
    - Lots of helpful features to help with debugging
    - Very useful in tracking undefined behavior

- Valgrind is a memory debugging tool
    - Checks for various memory errors
    - If you are running into odd behavior, running valgrind may point out the cause.

Segmentation Faults

- Causes of segmentation fault
  - Dereferencing uninitialized pointer
  - Null pointer
  - A previously freed pointer
  - Accessing end of an array
  - …
- gdb (GNU Debugger) is very helpful for identifying the source of a segmentation fault
    - backtrace

Other Essential gdb commands

- run <command_line_args>
- backtrace
- frame, up, down
- print <expression>
- quit
- breakpoints
    - (see next slide)

gdb reference card w/ commands & details on the course website.

gdb Breakpoints

- Usage:
  - break <function_name>
  - break <filename:line#>
- Can advance with:
  - continue
  - next
  - step
  - finish

Valgrind & Memory Errors

- Use of uninitialized memory
- Reading/writing memory after it has been freed – Dangling pointers
- Reading/writing to the end of malloc'd blocks
- Reading/writing to inappropriate areas on the stack
- Memory leaks where pointers to malloc'd blocks are lost

Valgrind is your friend!!

Structs and user defined types

Defining Structs

To define a struct, we use the **`struct`** statement.

A struct typically has a name (a tag), and one or more members.

The **`struct`** statement defines a new type.

```
struct fruit_st {
  char* name;
  int price_cents;
};
```

Initialising structs and changing field values

- By default the fields of a structure are public
- To change the field names or initialise their values we use the **dot** (.) operator or the **arrow operator (->)**

```
struct fruit_st fruit;
fruit.name = apple;
fruit.price_cents = 10
```

NB:the **arrow operator (->)** when you have a pointer to a struct

User Defined Types

The C Programming language provides the keyword **typedef**, which defines an alternate name for a type

```
typedef struct fruit_st {
  char* name;
  int price_cents;
} Fruit;
```

```
Fruit fruit;
fruit.name = apple;
...
```

No need for “struct” in type declaration

The Heap

Dynamically Allocated data

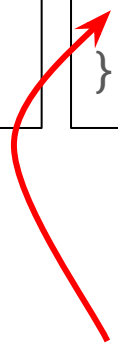
Dynamically allocated data is explicitly allocated and de-allocated by the program.

Dynamically allocated data persists after a function call

Dynamic vs automatic allocation

```
// dynamic allocation
int* foo() {
  int* x;
  x = malloc(sizeof(int));
  *x = 595;
  return x;
}
```

```
// “Automatic” Allocation
int* foo() {
  int *x;
  int n = 595;
  x = &n;
  return x;
}
```



x would be pointed to de-allocated memory.
“n” goes away when we return

User Defined Types (malloc)

```
Fruit* new_fruit = (Fruit*) malloc(sizeof(Fruit));
new_fruit->name = apple;
new_fruit->price_cents = 10
```

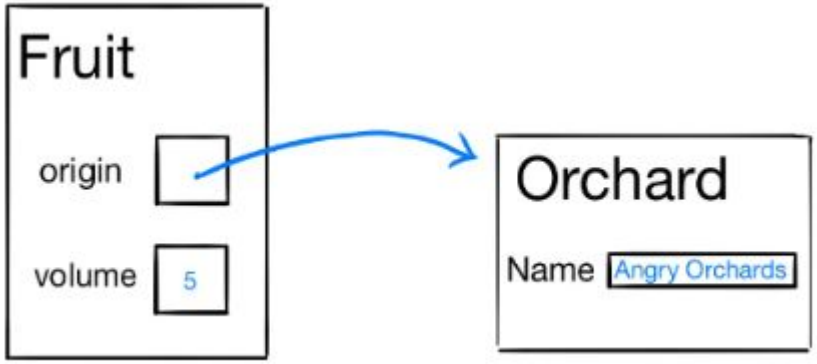
Pointers and Structs

Fruits & Orchards

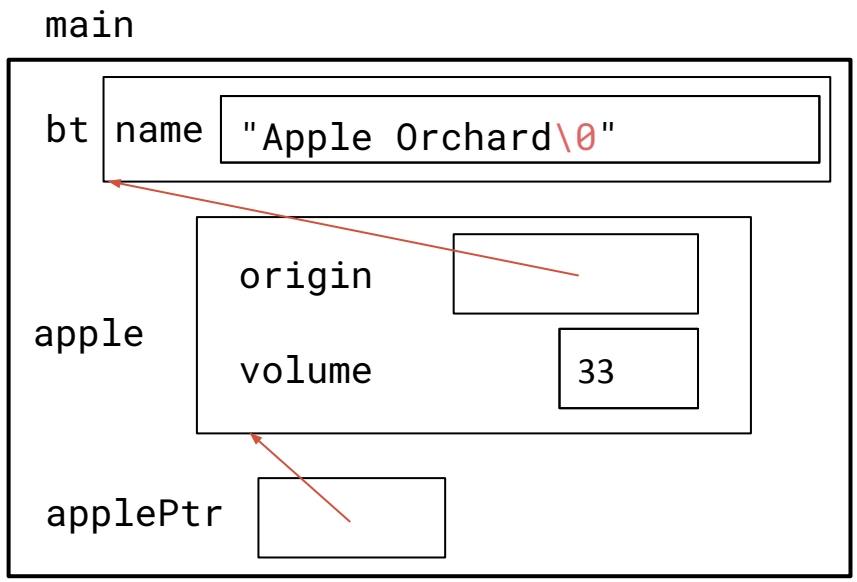
```
typedef struct fruit_st {
  OrchardPtr origin;
  int volume;
} Fruit;
```

```
typedef struct orchard_st {
  char name[20] ;
} Orchard, *OrchardPtr;
```

```
Orchard o;
o.name = "Angry Orchards";
Fruit f;
f.origin = &o;
f.volume = 5;
```



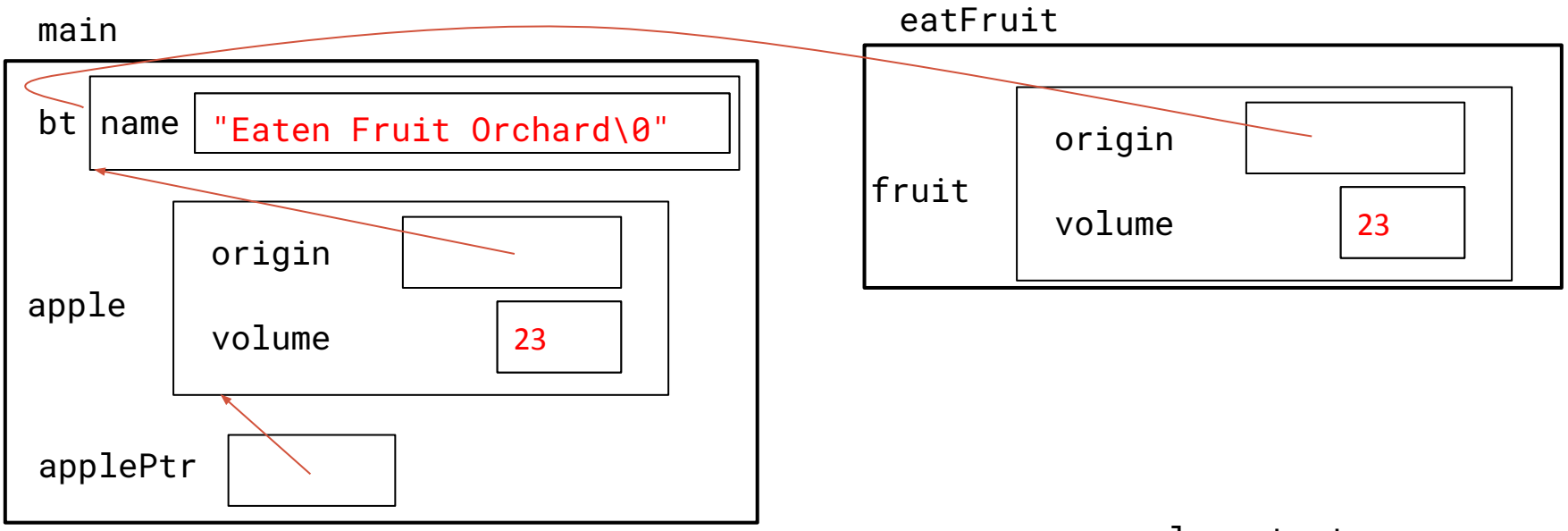
```
int main(int argc, char* argv[]) {
  Orchard bt;
  strcpy(bt.name, "Apple Orchard");

  Fruit apple;
  Fruit* applePtr = &apple;
  apple.origin = &bt;
  apple.volume = 33;
  applePtr->volume = apple.volume;

  printf("1. %d, %s \n",
    applePtr->volume,
    applePtr->origin->name);
  …
```



console output

```
1, 33, Apple Orchard
```

```
…
apple.volume = eatFruit(apple);
printf("2. %d, %s \n", applePtr->volume,
        applePtr->origin->name);
```

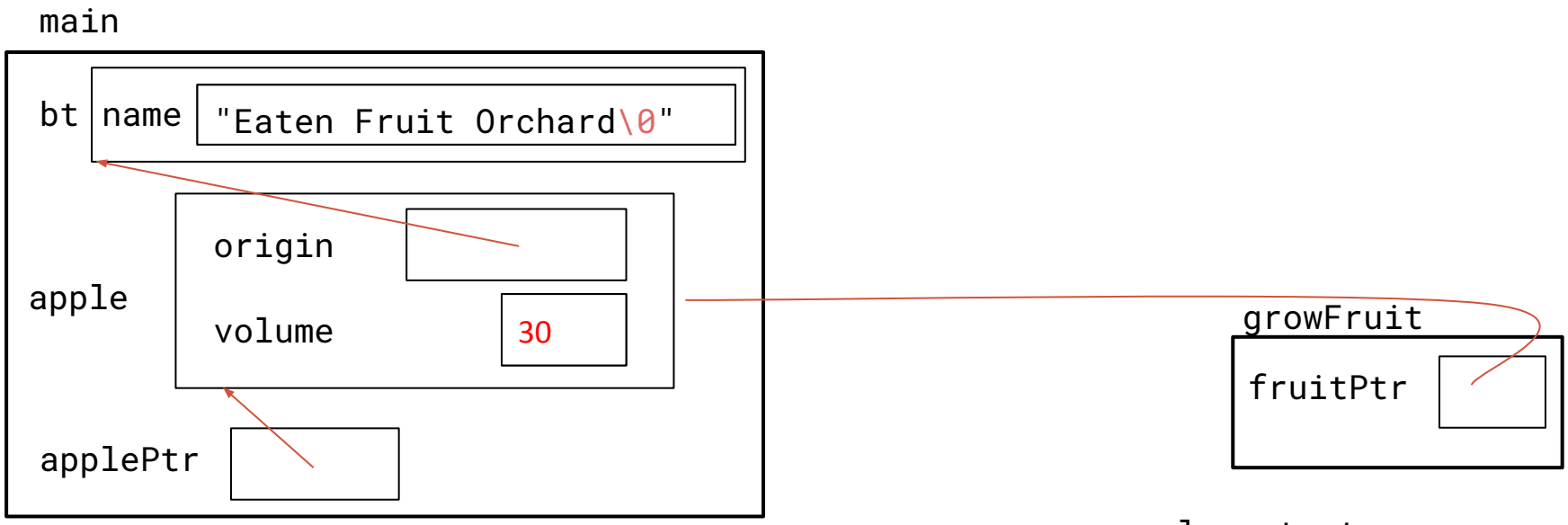



```
int eatFruit(Fruit fruit) {
  fruit.volume -= 10;
  strcpy(fruit.origin->name,
      "Eaten Fruit Orchard");
  return fruit.volume;
}
```

console output

```
1, 33, Apple Orchard
2, 23, Eaten Fruit Orchard
```

```
…
growFruit(applePtr);
printf("3. %d, %s \n", applePtr->volume,
    applePtr->origin->name);
```

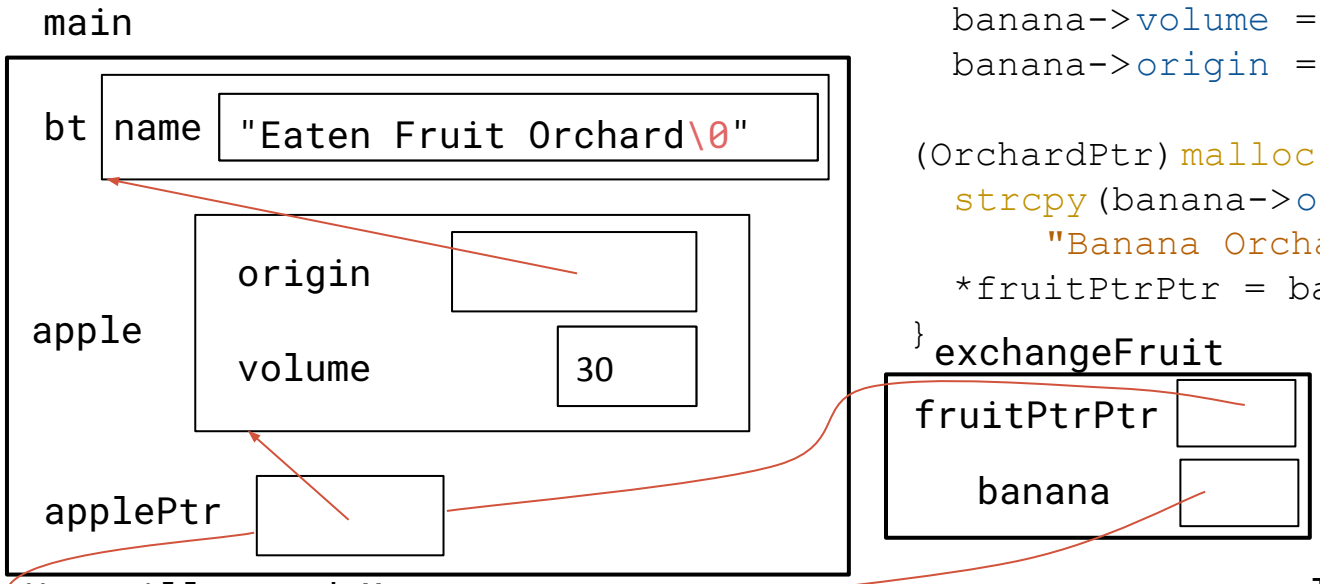



```
void growFruit(Fruit* fruitPtr) {
  fruitPtr->volume += 7;
}
```

console output

```
1, 33, Apple Orchard
2, 23, Eaten Fruit Orchard
3, 30, Eaten Fruit Orchard
```

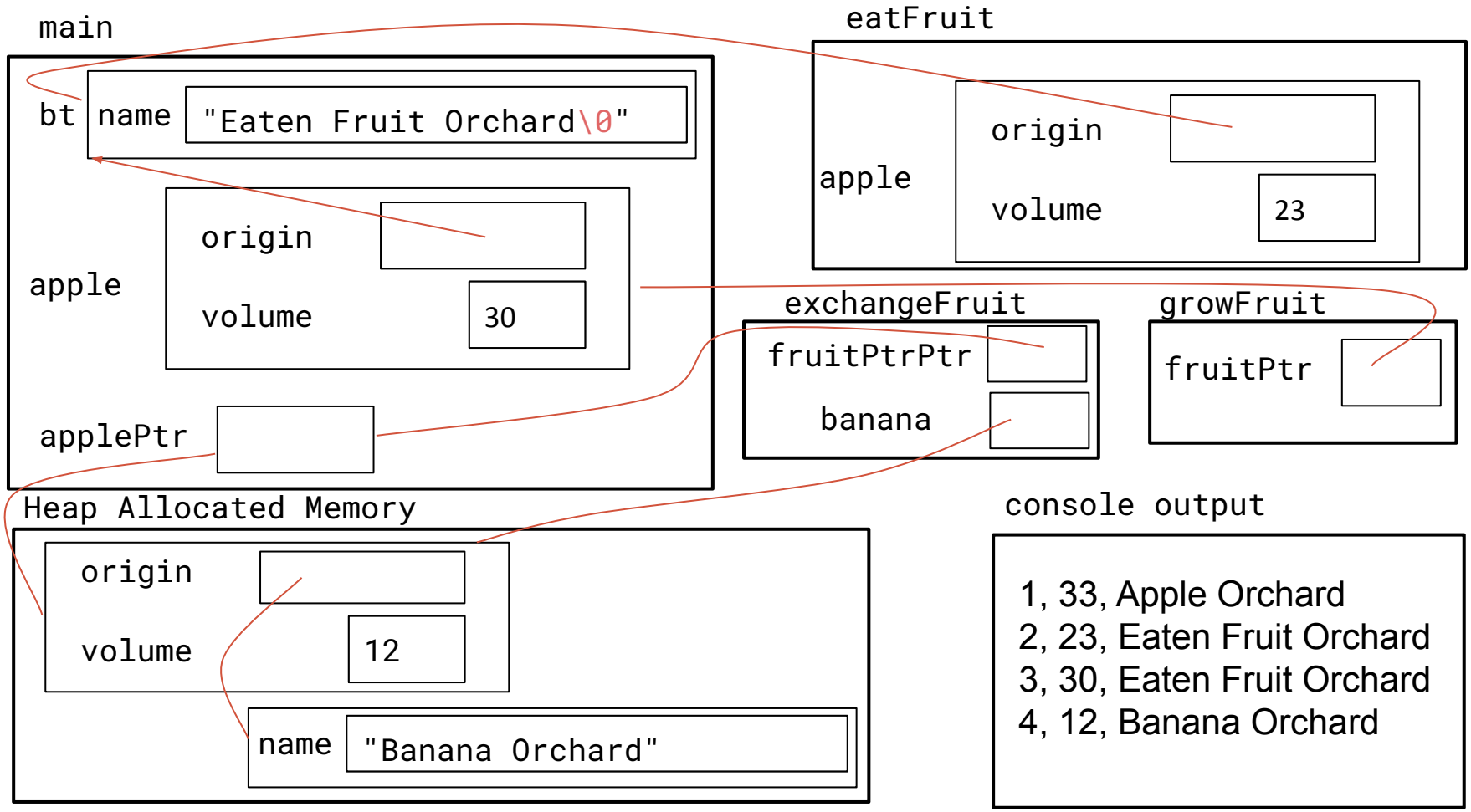
```
void exchangeFruit(Fruit** fruitPtrPtr) {
  Fruit *banana =
       (Fruit*)malloc(sizeof(Fruit));
  banana->volume = 12;
  banana->origin =

(OrchardPtr)malloc(sizeof(Orchard));
  strcpy(banana->origin->name,
       "Banana Orchard");
  *fruitPtrPtr = banana;
}
```



console output

1, 33, Apple Orchard
2, 23, Eaten Fruit Orchard
3, 30, Eaten Fruit Orchard
4, 12, Banana Orchard

```
exchangeFruit(&applePtr);
printf("4. %d, %s \n", applePtr->volume,
    applePtr->origin->name);
```


main
bt
name
"Eaten Fruit Orchard\0"
apple
origin
volume
30
applePtr
eatFruit
apple
origin
volume
23
exchangeFruit
fruitPtrPtr
banana
growFruit
fruitPtr
Heap Allocated Memory
origin
volume
12
name
"Banana Orchard"
console output
1, 33, Apple Orchard
2, 23, Eaten Fruit Orchard
3, 30, Eaten Fruit Orchard
4, 12, Banana Orchard